(4

JMIACS-TR-88-15
:S-TR-1991

February, 1988

Scheduling Tasks in a Real-Time System†

Prasad R. Chintamaneni and Xiaoping Yuan
Department of Computer Science
Satish K. Tripathi and Ashok K. Agrawala‡g
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

4

UMIACS–TR–88–15
CS–TR–1991

February, 1988

Scheduling Tasks in a Real–Time System†

Prasad R. Chintamaneni and Xiaoping Yuan
Department of Computer Science
Satish K. Tripathi and Ashok K. Agrawala‡g
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

ABSTRACT

A heuristic technique for dynamically scheduling tasks in a real–time system is described in this paper. Each task is defined by a being time, a deadline and a required computation time. The scheduling problem is to schedule every task successfully, i.e., each task should start after its begin time and complete its execution before its deadline. Two characteristics distinguish our task model and heuristic technique: First, a new task to be scheduled is allowed to modify a previous, partial schedule that is already feasible. Thus, the heuristic technique can be used either to build a feasible schedule from scratch or extend a partial, feasible schedule. Second, each task is bounded by a window of time defined by the *begin* and *end* time. The begin time is typically equal to or grater than the task arrival time. The end time is typically equal to or less than the task deadline. The additional constraint of starting a task after specified time instant is useful in task synchronization. Comparisons with known heuristic techniques for dynamically scheduling tasks in a real–time system are made via simulation. The results show that the proposed approach has a higher degree of success in obtaining a feasible schedule compared to the other approaches.

† This work is supported in part by contract No. N00014–87–K–0241 from the Office of Naval Research to the Department of Computer Science, University of Maryland at College Park.

‡g The authors would like to acknowledge the comments of Nan Muckenhim on the writeup of this paper.

Scheduling Tasks in a Real-Time System[1]

Prasad R Chintamaneni, Xiaoping Yuan, Satish K Tripathi, and Ashok K Agrawala[2]
Department of Computer Science and
Institute for Advanced Computer Studies,
University of Maryland,
College Park, MD 20742.

Abstract

A heuristic technique for dynamically scheduling tasks in a real-time system is described in this paper. Each task is defined by a begin time, a deadline and a required computation time. The scheduling problem is to schedule every task successfully, i.e., each task should start after its begin time and complete its execution before its deadline. Two characteristics distinguish our task model and heuristic technique: First, a new task to be scheduled is allowed to modify a previous, partial schedule that is already feasible. Thus, the heuristic technique can be used either to build a feasible schedule from scratch or extend a partial, feasible schedule. Second, each task is bounded by a window of time defined by the *begin* and *end* times. The begin time is typically equal to or greater than the task arrival time. The end time is typically equal to or less than the task deadline. The additional constraint of starting a task after specified time instant is useful in task synchronization. Comparisons with known heuristic techniques for dynamically scheduling tasks in a real-time system are made via simulation. The results show that the proposed approach has a higher degree of success in obtaining a feasible schedule compared to the other approaches.

1. Introduction

Process control, manufacturing systems, and nuclear power plants are some examples of applications involving distributed, real-time tasks. The real-time constraints in these applications are *hard* in that the tasks have to meet their execution deadlines. Satisfying the time requirements of the tasks in a system motivates the need for proper scheduling of system resources so that the timing behavior of the system is predictable and maintainable.

Different scheduling algorithms make different assumptions about the set of resources and the set of tasks in the system. For example, a static version of the scheduling problem involves the assumption of complete and prior knowledge of the task sets. Specifically, the arrival times, the deadlines, the worst-case computation times, the inter-task precedence constraints and the

[1]This work was supported in part by contract N00014-87-K-0241 from the Office of Naval Research to the Department of Computer Science, University of Maryland, College Park, MD 20842.

[2]The authors would like to acknowledge the comments of Nan Muckenhirn on the writeup of this paper.

resource requirements of each task are assumed to be known. In general, incorporating this information in scheduling is costly and inflexible. Moreover, the assumption of complete and prior knowledge, which implies a closed system, is questionable in actual practice. Another version of the scheduling problem involves the assumption that the tasks are typically compute bound. Specifically, the *cpu* is the only resource of contention considered at each node of the distributed system. All other resources are assumed to be contention-free. This assumption is not true of all real-time systems. In this paper, we address the question of scheduling for loosely coupled distributed systems where prior and complete knowledge of the task sets is not required and all system resources are likely candidates for contention.

We base our approach to scheduling in a loosely coupled distributed system on the one proposed in [14][16][18]. In this approach, each node of the distributed system contains a local *scheduler*, *bidder* and *dispatcher*. A task arrival at a node results in the invocation of the local scheduler. The job of the local scheduler is to decide whether the task can be *guaranteed* at the node locally. A guaranteed task is ensured to satisfy its time and resource constraints. In the event a task cannot be guaranteed locally, it is either terminated or sent to another node. The decision on where to send a non-terminated task is based on a *bidding* scheme[2]. Each bidder is responsible for: 1) sending out bids in response to requests from other nodes, 2) sending out requests for bids for a task that cannot be guaranteed locally, 3) evaluating bids from other nodes sent in response to requests from the local node, and 4) sending tasks that cannot be guaranteed locally to the best bidder. The actual scheduling of the guaranteed tasks at a node is performed by the local dispatcher. This approach to scheduling tasks in a real-time distributed system is flexible and maintainable. In this paper we assume the existence of, and use the functionality provided by, the local bidder and dispatcher. We concentrate on improving the behavior of the local scheduler and propose a heuristic scheduling technique towards this end.

The organization of the paper is as follows. Section 2 contains a brief review of the previous work related to this problem. The applicability of a specific scheduling technique depends on the model used to describe a task in a system. In section 3, we specify the task model used by our scheduler. The usefulness of a scheduling technique is determined by its ability to react to changes in system workload. Specifically, the algorithm must be applicable to scheduling on-line, batch, and periodic tasks. In section 4, we describe our heuristic scheduling technique and discuss its applicability. A simulation study comparing our scheduling technique with several others is presented in section 5. It is shown that our technique performs better than the others considered. While the optimality of a scheduling algorithm is important, the complexity of the algorithm is also an important aspect in real-time scheduling. Section 6 includes an analysis of the complexity and optimality of all the scheduling techniques considered in section 5. Our conclusions are summarized in section 7.

2. Previous Work

Real-time scheduling research dates back to the early 1970's. A deadline-driven algorithm for two processors is given in [3], and the authors introduced the concept of modified deadlines by taking the precedence relation among the tasks into consideration. They proved that the problem of minimizing the number of deadline-missing tasks is NP-complete. Liu and Layland [8] developed an optimal, static, priority-driven scheduling algorithm that attained the upper bound of processor utilization for a periodic task set. They also showed that full processor utilization could be obtained by using a dynamic priority assignment, such as the earliest-deadline-first algorithm. Variations of the deadline driven scheduling algorithms are found in [4, 11, 15]. Several authors [11, 12, 15] have proposed data-driven graph models for real-time scheduling. A branch and bound algorithm applied to allocate resources and to schedule tasks in a multiple processor system is discussed in [9]. Scheduling algorithms dealing with the guaranteed response times of tasks in a local processor or in a distributed real-time environment are discussed in [5, 6, 17]. Task clustering scheduling algorithms, intended to reduce scheduling and communication overheads by clustering related tasks together, have been studied in [1, 2]. A variety of heuristics in scheduling, incorporating multiple resource requirements, preemptions and sharing, have been studied in [16, 18, 19]. Scheduling strategies for handling both periodic and aperiodic tasks are discussed in [7].

3. System and Task Model

Let R be the set for resources of the entire distributed system: R $= \{R_1, R_2, \dots, R_r\}$. Each node in the system is considered to have a single instance of one or more of the resources in R. The set of resources is partitioned into two groups: *active* resources and *passive* resources. Every task in the system has to use at least one active and one passive resource for its computation. A passive resource can be used either in *exclusive* mode or *shared* mode. The former refers to exclusive access and the latter refers to shared access of the specific, passive resource. We denote a resource i as R_i if it is active, R_i' if it is passive, and $\overline{R_i'}$ if it is passive and shared.

Each task T_i in the system is defined by a time constraint, TC_i : $TC_i = \{B_i, C_i, D_i\}$. B_i indicates the *begin* time of the task. Typically, it is equal to or greater than the arrival time of the task. In this paper, we assume that the *begin* time of a task is to its *arrival* time. C_i specifies the task *computation* time and D_i specifies the task *end* time (Without loss of generality, we assume the *end* time of a task is equal to its *deadline* time). For successful scheduling, the task has to start execution after B_i and complete execution before D_i.

Additionally, each task T_i is associated with a resource requirement RR_i : $RR_i \epsilon R$. As mentioned earlier, for every task the resource requirement must include at least one active resource and one passive resource. Passive resources can be used in shared or exclusive modes.

4. The Scheduling Algorithm

The different data structures used by the algorithm are described in section 4.1. In section 4.2 the scheduling algorithm is described and in section 4.3 the applicability of the algorithm is discussed.

4.1. Data Structures

4.1.1. Earliest Available Times

A doubly linked list of structures named EAT to indicate the *earliest available times* of resources is maintained. Each EAT structure in the linked list corresponds to a successfully scheduled task in the system, and is defined as a vector:

$$EAT_j = (EAT_{j1}, EAT_{j2}, \ldots, EAT_{jr})$$

The j^{th} EAT structure in the linked list, EAT_j, defines the earliest times each resource in the system becomes available, subject to the constraint that all the tasks 1 through j in the linked list are successfully scheduled. Specifically EAT_{ji} is the earliest time when resource R_i will become available, after all the tasks 1 through j in the linked list are successfully scheduled. Each time the partial schedule is extended, one or more EAT structures will be updated taking into account the new task's resource requirements, completion time and inserted position in the schedule.

Example: Assume that we have six active resources $R_1, R_2, .., R_6$. Consider the problem of scheduling three tasks with the following descriptions:

$$\text{T1: } TC_1 = \{0, 10, 12\} \quad RR_1 = \{R_2, R_3, R_5\}.$$

$$\text{T2: } TC_2 = \{0, 20, 40\} \quad RR_2 = \{R_1, R_2, R_6\}.$$

$$\text{T3: } TC_3 = \{10, 9, 20\} \quad RR_3 = \{R_2, R_3, R_4\}.$$

After the first task is included in the schedule, the corresponding EAT structure is:

$$EAT_1 = (EAT_{11}, EAT_{12}, EAT_{13}, EAT_{14}, EAT_{15}, EAT_{16}) = (0,\ 10,\ 10,\ 0,\ 10,\ 0).$$

The earliest time resources R_2, R_3 and R_5 will be available for other tasks is 10 time units. All the remaining resources are free for use from time 0.

For all subsequent tasks (j > 1), the earliest starting time, EST_j, of a specific task, T_j, is given by

$$EST_j = \text{MAX}(EAT_{(j-1)i})$$

for all i (i=1,2 ...,r) satisfying the condition that R_i is required by T_j.

The task can be successfully scheduled if

$$\text{MAX}(B_j, EST_j) + C_j <= D_j.$$

The corresponding EAT structure is defined by

$$EAT_{ji} = EAT_{(j-1)i} \text{ if } R_i \text{ is not required by } T_j \text{ and,}$$

$$EAT_{ji} = \text{MAX}(B_j, EST_j) + C_j \text{ if } R_j \text{ is required by } T_j.$$

Thus, if the second task is scheduled next, the corresponding EAT would be

$$EAT_2 = (30, 30, 10, 0, 10, 30).$$

The third task cannot be scheduled at the end of the partial schedule, since the deadline constraint will be violated. It can, however, be inserted between the first two tasks in the linked list. In that case, the EAT linked list will be as follows:

$$EAT_1 = (0,\ 10,\ 10,\ 0,\ 10,\ 0),$$

$$EAT_2 = (0,\ 19,\ 19,\ 19,\ 10,\ 0),$$

$$EAT_3 = (39,\ 39,\ 19,\ 19,\ 10,\ 39).$$

The earliest starting times of the three tasks in the linked list are therefore $EST_1 = 0$, $EST_2 = 10$, and $EST_3 = 19$.

4.1.2. Latest Needed Times

A doubly linked list of structures named LNT to indicate the *latest needed times* of resources is also maintained by the algorithm. Each LNT structure in the linked list corresponds to a successfully scheduled task in the system, and is defined as a vector:

$$LNT_j = (LNT_{j1},\ LNT_{j2},\ \ldots,\ LNT_{jr})$$

The j^{th} LNT structure in the linked list, LNT_j, defines the latest time all the resources are needed to satisfy the constraint of successfully scheduling the tasks j through n in the linked list, where n corresponds to the last task in the linked list of scheduled tasks. Specifically, LNT_{ji} is the latest time when resource R_i is needed, subject to the constraint that all tasks j through n in the linked list can be successfully scheduled. Each time a partial schedule is extended, one or more LNT structures will be updated taking into account the new task's resource requirements, completion time, and inserted position in the schedule.

Consider the same example as before. The last task on the linked list has a single constraint to satisfy, i.e., it should complete before its deadline. If a resource is used by this task, the latest time it is needed is equal to the difference of its deadline time and its computation time. If a resource is not used by this task, the latest time it is needed is effectively infinity. Thus LNT_3 is given by

$$LNT_3 = (20,\ 20,\ \infty,\ 20,\ \infty,\ 20).$$

All other tasks in the list have to satisfy two constraints, i.e., they should complete before their deadlines and they should not violate the schedulability of the tasks succeeding them in the list. For all tasks ($j < n$), the latest possible time a specific task can be completed, LFT_j, is given by

$$LFT_j = \mathrm{MIN}(LNT_{(j+1)i}),$$

for all i (i = 1, . . . , r), provided R_i is used by T_j.

The corresponding LNT structure is defined by

$$LNT_{ji} = LNT_{(j+1)i} \text{ if } R_i \text{ is not used by } T_j \text{ and,}$$

$$LNT_{ji} = \mathrm{MIN}(D_j, LFT_j) - C_j \text{ if } R_j \text{ is used by } T_j.$$

Specifically, the LNT linked list in the example will be as follows:

$$LNT_3 = (20,\ 20,\ \infty,\ 20,\ \infty,\ 20),$$

$$LNT_2 = (20,\ 11,\ 11,\ 11,\ \infty,\ 20),$$

$$LNT_1 = (20,\ 1,\ 1,\ 11,\ 1,\ 20).$$

The latest starting time of a task, LST_j, is thus equal to the latest finish time of the task preceding it.

$$LST_j = LFT_{(j-1)}.$$

The latest starting times of the three tasks in the linked list are $LST_1 = 1$, $LST_2 = 11$, and $LST_3 = 20$.

4.1.3. Task Information

The algorithm maintains a doubly linked list of structures named TASK to hold information about the tasks scheduled in the system. Each TASK structure in the linked list corresponds to a successfully scheduled task in the system, and is defined as a vector:

$$TASK_j = (B_j, C_j, D_j, S_j, F_j),$$

where B_j is the begin time (typically the arrival time), C_j the computation time, D_j the deadline, S_j the start time, and F_j the finish time of the j^{th} task in the linked list. There is a one-to-one correspondence between the three linked lists maintained by the scheduler.

4.2. Algorithm description

We employ a simple strategy, *compress and insert*, to determine the schedulability of a task at a node. The basic idea is to determine if a new task can be *inserted* at *any* position in the partial, feasible schedule so that the feasibility of the previous schedule is retained. This differs from the majority of the previous approaches in determining whether a new task can be *added* to the *end* of the partial, feasible schedule so that the resulting schedule is still feasible.

The data structures described in the previous section are used by the heuristic described below. Given a task T {i.e., its time constraint and resource requirement} and a partial, feasible schedule the heuristic, in its simplest form, attempts to schedule T as follows:

1. *Compress* the list of tasks already scheduled. The effect of task compression is to determine the earliest time each task in the list can start execution. The procedure works from the beginning of the list of scheduled tasks towards the end of the list, updating EAT. Specifically, EAT_i will yield the earliest times each of the resources will be free, after the tasks 1 through i are successfully scheduled.

2. *Expand* the list of tasks already scheduled. The effect of task expansion is to determine the latest time each task in the list can start execution. The procedure works from the end of the list of scheduled tasks towards the beginning of the list, updating LNT. Specifically, LNT_i will yield the latest times each of the resources will be needed, so that the tasks i through the end of the list can be successfully scheduled.

3. *Check* if T can be inserted at each possible position in the list of scheduled tasks. With n tasks in the partial schedule, there are $n+1$ possible positions for T. Specifically, T can be inserted in the list at position i if the window of time defined by $EAT_{(i-1)}$ and LNT_i is large enough to satisfy its time constraints and resource requirements. The procedure works backwards through the list of scheduled tasks to the beginning of the list. The three linked lists are updated in the event of an insertion. A *failure* is returned otherwise.

The pseudo code for the proposed algorithm is given in Fig. 1.

```
Schedule(task, task_list, eat_list, lnt_list)
struct TASK *task ;       /* Task to be scheduled */
struct TASK *task_list ;  /* List of tasks already scheduled */
struct EAT *eat_list ;    /* EAT structures corresponding to task list */
struct LNT *lnt_list ;    /* LNT structures corresponding to task list */
{

        /* Find the earliest starting times of all the
          tasks already scheduled */
        Compress(EAT_List);

        /* Check if the new task can be inserted at the end
          of the partial schedule */
        If (Insertable(task, end_of_eat_list + 1))
         {
          Insert (task, end_of_eat_list + 1, eat_list, task_list, lnt_list) ;
          return (SUCCESS) ;
         }

        /* Check if the task can be inserted at each internal
        slot in the partial schedule. Start from the end */
        for ( j = end_of_lnt_list ; j >= 0 ; j--)
         {
          expand (lnt_list, j) ;  /* expand j^th task */
          If (Insertable(task, j))
            {
              Insert(TASK, j, eat_list, task_list, lnt_list) ;
              return (SUCCESS);
            }
         }

        /* Task cannot be inserted at any position */
        return (FAILURE) ;
}
```

Figure 1: Pseudo Code: Compress and Insert Heuristic.

Given a task to be scheduled, the algorithm in Figure 1 is the simplest implementation of the heuristic. In the worst case, two passes are made through the list of scheduled tasks: one for *compressing* and the other for *expanding*. However, notice that whenever a task is inserted at position j, all the tasks from 1 through j are already compressed and all the tasks from $j+1$ through the end of the schedule are already expanded. With two pointers, in the average case, one pass through the list of scheduled tasks is sufficient to schedule a new task.

4.3. Algorithm Applicability

Typical categories of tasks in a real-time, distributed system include *sporadic* tasks, *periodic* tasks, task sets with *precedence constraints* and task sets characterized by *bulk arrivals*. Sporadic tasks are characterized by one-shot executions. Periodic tasks are characterized by repeated executions with a pre-determined time interval between any two executions. Tasks with precedence constraints impose additional temporal constraints on the order of task execution. Bulk arrivals specify a group of tasks arriving at a node at the same time.

The *compress and insert* algorithm is applicable in the cases of tasks that arrive either sporadically or in bulk. Given a set of tasks to be scheduled, the algorithm determines the scheduling of tasks one at a time. The algorithm makes no distinction between building a schedule from scratch or extending a partial, feasible schedule.

The algorithm is also applicable in scheduling task sets with precedence constraints. Notice that the scheduling not only guarantees that a task completes execution *before* its deadline, but also guarantees that it starts execution *after* its begin time. The basic strategy in scheduling tasks with precedence constraints is to identify or fix the begin times of the individual subtasks in the precedence set. The individual subtasks can then be scheduled by the algorithm independently, and the precedence constraints are still not violated. To handle periodic tasks, the scheduling of each instance of the periodic task needs to be followed by the inclusion of a new, succeeding instance of the task in the list of unscheduled tasks. We are in the process of including these features in the testbed.

5. Simulation Results and Observations

Ten different heuristics were chosen for comparison. Each heuristic tested the schedulability of a specific task by checking that the task could start *after* its begin time and complete its execution *before* its deadline, subject to satisfying its resource constraints. If all the tasks in a given set are schedulable, the heuristic is said to yield a feasible schedule. A brief description of each of the heuristics is given below:

[H1] **Earliest Deadline:** Given a set of tasks to be scheduled, the algorithm selects the task with the earliest deadline to be added to the end of the partial, feasible schedule.

[H2] **Least Laxity:** Given a set of tasks to be scheduled, the algorithm selects the task with the least laxity to be added to the end of the partial, feasible schedule.

[H3] **Sorted Earliest Deadline:** Given a set of tasks to be scheduled, the algorithm selects the task with the earliest begin time to be added to the end of the partial, feasible schedule. If there is more than one such task, the one with the earliest deadline is chosen.

[H4] **Sorted Least Laxity:** Given a set of tasks to be scheduled, the algorithm selects the task with the earliest begin time to be added to the end of the partial, feasible schedule. If there is more than one such task, the one with the least laxity is chosen.

[H5] **Compress and Insert:** Given a set of tasks to be scheduled, the algorithm selects a random task to be inserted in the partial, feasible schedule, according to the heuristic described in the previous section.

[H6] **Compress and Insert + Deadline:** Given a set of tasks to be scheduled, the algorithm selects the task with the earliest deadline to be inserted in the partial, feasible schedule, according to the heuristic described in the previous section.

[H7] **Compress and Insert + Laxity:** Given a set of tasks to be scheduled, the algorithm selects the task with the least laxity to be inserted in the partial, feasible schedule, according to the heuristic described in the previous section.

[H8] **Umass:** Given a set of tasks to be scheduled, the algorithm selects a task heuristically (see [2] for details of the heuristic function employed) to be added to the end of the partial, feasible schedule.

[H9] **Compress and Insert + Sorted Laxity:** Given a set of tasks to be scheduled, the algorithm selects the task with the earliest begin time to be inserted in the partial, feasible schedule, according to the heuristic described in the previous section. If there is more than one such task, the one with the least laxity is selected.

[H10] **Compress and Insert + Sorted Deadline:** Given a set of tasks to be scheduled, the algorithm selects the task with the earliest begin time to be inserted in the partial, feasible schedule, according to the heuristic described in the previous section. If there is more than one such task, the one with the earliest deadline is selected.

A task generator was built to generate a feasible schedule of a set of tasks for a given system configuration (i.e., number of active and passive resources). The parameters of the task generator include the number of tasks in the task set, the number of active resources in the system, and the number of passive resources in the system. Additional control parameters specified whether sharing of resources was allowed or prohibited, and whether the arrival times of the tasks were zero (begin times = 0 implies bulk arrivals) or whether the task arrival times were non-zero (i.e., sporadic arrivals). Task-specific parameters in the generator included the *minimum computation time*, the *maximum computation time*, the *maximum laxity*, and a *control parameter*, c_p. Each task generated had a computation time that was a random variable drawn from a uniform distribution bounded by the minimum and maximum computation times.

The *laxity* of a task is defined as the difference between the deadline time of the task and the sum of the task arrival time and computation time. (i.e., laxity = D_t - B_t - C_t.) Each task generated had a laxity that was a random variable drawn from a uniform distribution bounded by zero and the maximum laxity. Tasks with arrival time zero however had an effective laxity greater than the value of the specified laxity parameter. The control parameter, c_p, defines the ratio between the *begin-laxity* and the *end-laxity*. The begin-laxity is defined as the difference between the task start time and the task begin time in the feasible schedule generated by the task generator. The end-laxity is defined as the difference between the task deadline time and the task finish time in the feasible schedule generated by the task generator.

Fo. he simulations, 100 task sets were generated for each experiment. Each task set was comprised of 10 tasks. Three factors were chosen for experimentation: the number of resources in the system, the *maximum laxity* of the tasks in the task set, and, c_p, the ratio between the *begin-*

laxity and the *end-laxity* of the tasks in the system.

For each factor, four different sets of experiments were conducted:

[1] NS, ATZ: No sharing of resources and arrival times of tasks zero.

[2] S, ATZ: Sharing of resources allowed and arrival times of tasks zero.

[3] NS, ATNZ: No sharing of resources and arrival times of tasks non-zero.

[4] S, ATNZ: Sharing of resources allowed and arrival times of tasks non-zero.

The baseline parameters for the task generator were:

Number of Active Resources	:	2
Number of Passive Resources	:	4
Minimum Computation Time	:	1
Maximum Computation Time	:	10
Maximum Laxity	:	20
Control Parameter	:	0.5

The simulation results are presented in Tables 1.1 -- 3.4. The results show the percentages of task sets for which feasible schedules were found by each heuristic.

Res	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
(2,4)	90	87	90	87	39	96	97	97	97	96
(3,5)	89	79	89	79	48	98	97	95	97	98
(4,6)	88	83	88	83	53	97	100	95	100	97
(5,7)	87	78	87	78	53	97	98	91	98	97
(6,8)	50	78	50	78	54	95	97	88	97	95

Table 1.1: Variation of the number of resources, NS, ATZ

Res	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
(2,4)	88	84	88	84	36	96	98	95	98	96
(3,5)	87	82	87	82	58	95	96	94	96	95
(4,6)	90	79	90	79	65	97	98	94	98	97
(5,7)	89	88	89	88	66	97	100	94	100	97
(6,8)	89	78	89	78	61	98	95	88	95	98

Table 1.2: Variation of the number of resources, S, ATZ

Res	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
(2,4)	18	0	44	44	53	52	62	74	94	95
(3,5)	20	0	46	45	52	55	64	64	96	95
(4,6)	24	0	38	40	59	59	65	65	93	89
(5,7)	33	0	44	42	75	75	66	74	90	91
(6,8)	32	0	50	50	46	76	63	70	97	97

Table 1.3: Variation of the number of resources, NS, ATNZ

Res	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
(3,5)	0	41	41	67	47	67	24	66	89	85
(4,6)	0	48	47	66	50	66	30	58	88	88
(5,7)	0	49	49	65	50	61	22	55	95	94
(6,8)	1	63	62	68	50	61	25	61	91	90

Table 1.4: Variation of the number of resources, S, ATNZ

Lax	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
3	96	83	96	83	11	98	100	38	100	98
6	81	69	81	69	12	90	97	60	97	90
9	79	59	79	59	16	89	97	71	97	89
12	80	67	80	67	23	91	93	79	93	91
15	85	75	85	75	32	95	99	96	99	95
18	89	83	89	83	39	98	99	95	99	98
25	97	89	97	89	51	100	99	97	99	100

Table 2.1: Variation of Laxity, NS, ATZ

Lax	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
3	92	79	92	79	14	95	100	38	100	95
6	87	72	87	72	14	93	98	61	98	93
9	80	59	80	59	22	90	96	78	96	90
12	84	60	84	60	26	93	96	86	96	93
15	86	69	86	69	31	97	96	91	96	97
18	87	80	87	80	33	97	97	95	97	97
25	95	80	95	80	49	99	100	96	100	99

Table 2.2: Variation of Laxity, S, ATZ

Lax	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
3	85	0	99	98	65	99	61	91	99	99
6	62	0	88	89	63	94	63	90	99	98
9	45	0	74	74	62	79	63	79	98	98
12	28	0	62	61	55	73	61	75	97	97
15	26	0	54	51	55	68	65	69	95	96
18	25	0	45	45	53	63	62	69	95	96
25	20	0	32	32	50	60	63	63	85	90

Table 2.3: Variation of Laxity, NS, ATNZ

Lax	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
3	78	0	99	96	60	97	63	79	99	99
6	63	0	89	84	57	96	69	72	98	98
9	40	0	77	70	53	82	65	72	96	97
12	30	0	68	65	45	71	62	68	95	95
15	30	0	53	53	41	63	56	61	96	93
18	26	0	45	44	40	53	59	55	96	95
25	28	0	28	28	43	63	66	64	95	92

Table 2.4: Variation of Laxity, S, ATNZ

c_p	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
0.1	90	87	90	87	39	96	97	97	97	96

Table 3.1: Variation of c_p , NS, ATZ

c_p	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
0.1	88	84	88	84	36	96	98	95	98	96

Table 3.2: Variation of c_p , S, ATZ

c_p	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
0.1	16	0	97	97	60	63	68	71	99	99
0.2	18	0	82	81	60	60	66	64	99	99
0.3	16	0	64	63	59	58	66	63	96	97
0.4	14	0	51	51	59	53	63	68	94	95
0.5	18	0	44	44	53	52	62	74	94	95
0.6	20	0	28	28	50	59	55	64	94	96
0.7	21	0	16	16	41	64	54	66	88	90
0.8	28	0	14	14	39	68	57	50	86	88
0.9	28	0	12	11	36	68	54	48	80	82

Table 3.3: Variation of c_p, NS, ATNZ

c_p	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
0.1	25	0	96	95	47	60	67	64	98	98
0.2	25	0	85	84	46	64	67	68	98	98
0.3	25	0	60	65	49	59	65	60	98	97
0.4	25	0	51	50	46	56	63	60	97	95
0.5	23	0	43	41	44	57	61	56	94	93
0.6	31	0	22	22	37	60	55	50	91	88
0.7	27	0	16	15	31	58	50	48	85	85
0.8	30	0	10	9	23	61	57	38	81	80
0.9	35	0	10	8	27	70	54	32	78	76

Table 3.4: Variation of c_p, S, ATNZ

Tables 1.1 - 1.4 present the percentages of feasible schedules found by each of the ten heuristics, when the number of resources in the system was varied. The column entry for the number of resources is written as (a, b) where a denotes the number of active resources in the system and b denotes the number of passive resources in the system. Tables 2.1 - 2.4 present the percentages of feasible schedules found by each of the ten heuristics, when the laxity is varied. In the experiments with tasks having a non-zero arrival time, the laxity parameter specifies the maximum difference between the task *deadline* and the sum of the task *begin time* and task *computation* time (i.e., laxity $= D_t - B_t - C_t$). In the experiments with tasks having an arrival time equal to zero, the laxity parameter specifies the maximum *end-laxity* of the tasks in the generated schedule by the task generator. The effective laxity in these experiments could therefore be greater than the specified laxity parameter. Tables 3.1 - 3.4 present the percentages of feasible schedules found by each of the ten heuristics, when the control parameter c_p is varied. Since c_p is the ratio between the task *begin_laxity* and the task *end_laxity*, it can only be varied for tasks with non-zero arrival times.

The following observations can be made from the different experiments conducted:

[a] Heuristics H9 and H10 perform consistently better than all the others considered.

[b] The sorted version of a heuristic performs better than its unsorted counterpart (e.g., H3 is better than H1, H4 is better than H2, H9 is better than H7, and H10 is better than H6.)

[c] All heuristics behave better with a zero arrival time than with a non-zero arrival time, as the number of resources in the system varies. The reason, as mentioned earlier, is that the effective laxity with a zero arrival time is much greater than with a non-zero arrival time. The heuristics have a better performance with a non-zero arrival time for low values of laxity. With high values of laxity, however, the trend reverses.

[d] The performance of the heuristics shows no consistent and significant patterns of behavior when experiments with sharing are compared to experiments with no sharing.

[e] The performance of the heuristics shows no consistent and significant patterns of behavior when the number of resources are changed in the system.

[f] The performance of the heuristics varies with laxity. The maximum baseline task computation time in these experiments was 10 time units. All the heuristics display a certain monotonic behavior till the maximum laxity equals the maximum computation time. They show a different monotonic behavior for values of maximum laxity greater than the maximum computation time. As expected, the ratio of the maximum computation time and maximum laxity is an important characterization of a real-time workload.

[g] The performance of the heuristics with the variation of c_p shows a declining trend behavior. Many of the heuristics are based on the task arrival time for selecting the next task to be scheduled. As the control parameter is increased, the *front_laxity* is increased, reducing the importance of task arrival time. This explains why the heuristics perform worse with increas-

ing values of c_p .

[h] When the arrival times of the tasks equal zero, the sorted and unsorted versions of all the heuristics yield the same results (i.e., H1 = H3, H2 = H4, H7 = H9, H6 = H10.)

6. Complexity Analysis

There is an obvious tradeoff between the performance of a scheduling algorithm and its complexity. The complexities of heuristics considered in the previous section are discussed below.

The *earliest deadline* (*least laxity*) algorithm selects the task with the closest deadline (least laxity), to be added to the end of the partial, feasible schedule. Given n jobs to be scheduled, sorting by deadline (laxity) is necessary. Thus the complexity of earliest deadline (least laxity) algorithm is O(nlogn). Given a single task to be added to the schedule, one check is sufficient to decide whether the time and resource constraints of the new task are satisfied.

The *sorted earliest deadline* and the *sorted least laxity* algorithms involve an additional sorting of the n tasks on the arrival times. Hence, their complexities are also O(nlogn). In addition, given a single task to be added to the schedule, one check is sufficient to decide whether the time and resource constraints of the new task are satisfied.

The *compress and insert algorithm* involves selection of a random task to be inserted in the partial, feasible schedule. Given a single task to be inserted in the schedule, in the worst case, if there are k tasks already scheduled, k compress operations, k expand operations and $k+1$ check operations need to be performed. For n tasks, the complexity is therefore $O(n^2)$.

The complexity of the algorithm *compress and insert + deadline* is also equal to $O(n^2)$, since an additional *nlogn* sort is the only additional requirement. The same is true for the algorithm *compress and insert + laxity.*

The *Umass* algorithm uses a set of heuristics to select the best task to be added to the end of a partial, feasible schedule. To select this task, all the tasks that remain to be scheduled are examined. As such, with k tasks to be scheduled, k operations are needed to select the next task to be scheduled. For n tasks, the complexity is therefore $O(n^2)$. However, for each selected task, a single check is sufficient to decide whether its time and resource constraints are satisfied.

The last two heuristics, H9 and H10 are again of complexity $O(n^2)$, involving additional sorts of complexity *nlogn*. As expected, the $O(n^2)$ algorithms perform better than the O(nlogn) algorithms, and among the $O(n^2)$ algorithms, H9 and H10 exhibit the best performance.

7. Conclusions

A major problem in finding optimal and cost-effective real-time scheduling algorithms is that resources are committed to the tasks that are scheduled first. The strategies typically employed in the past have attempted to minimize this problem by a judicious selection of the tasks to be scheduled (i.e., added to the end of the partial, feasible schedule). Thus, the major issue addressed is the selection of a task that can be scheduled next. Once a task is scheduled, its claim on the resources allocated are fixed forever. Notice also that when a task is scheduled, certain resources in the system could be constrained to be idle.

The approach presented in this paper tries to separate the scheduling problem into two parts: 1) Selection of the next task to be scheduled, given a set of tasks and 2) placement of the selected task into a feasible schedule. The problem of committing resources to tasks scheduled early still exists, in that the partial order of the tasks already scheduled remains the same after a new task is included in the schedule. However, the tasks are allowed to move within a window of time which is delimited on one hand by their resource and time constraints and on the other hand by the partial order of the tasks already scheduled. To incorporate this added flexibility, we pay a price in that the complexity of the algorithm is $O(n^2)$.

The performance of the proposed algorithms, H9 and H10, are superior to all the algorithms considered. The average success percentage of H9, over all the sets of experiments, is 95% while that of H10 is 93%. The same algorithms can be used for building a schedule from scratch or for extending a partial, feasible schedule.

Four problems can be identified for future work. First, one could implement the algorithm for handling both periodic jobs and jobs with precedence constraints. Second, one could investigate preempt'‐'e alternatives for scheduling. Third, one could account for the computation time for scheduling and guaranteeing tasks in the system. Finally, since interrupts cannot be avoided in any operating system, one could investigate strategies that account for the time to service interrupts in the scheduling of resources.

References

[1] Cheng, S., J. A. Stankovic, K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems," *Proc. IEEE Real-Time Syst. Symp.*, Dec.1986.

[2] Efe, K., "Heuristic Models of Task assignment scheduling in Distributed Systems," *IEEE Computer*, June 1982.

[3] Garey, M.R., and D. S. Johnson, "Scheduling Tasks with Nonuniform Deadlines on Two Processors," *JACM*, Vol. 23, 1976.

[4] Gonzalez, C., K. Y. Jo, "Scheduling with Deadline Requirements," *Proc. of 1985 ACM annual conference (Denver, Colorado)*, Oct. 1985.

[5] Leinbaugh, D.W. and M. R. Yemini, "Guaranteed Response Times in a Distributed Hard Real-Time Environment," *Proc. Real-Time Systems Symp.*, 1982.

[6] Leinbaugh, D., "Guaranteed Response Time In a Hard-Real-Time Environment," *IEEE Trans. Soft. Eng.*, Vol. SE-6, No.1, Jan. 1980.

[7] Lehoczky, J., L. Sha, J. Strosnider, "Enchanced Aperiodic Responsiveness in Hard Real-Time Environments," *Proc. IEEE Real-Time Syst. Symp.*, Dec. 1987.

[8] Liu, C. L. and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *JACM*, Vol. 20, No. 1, 1973.

[9] Ma, R. P., E. Lee, and M. Tsuchiya, "A Task Allocation Model for Distributed Computer Systems," *IEEE Transactions on Computers," Vol. C-31, No. 1, 1982.*

[10] Mok, A.K., and Dertouzos, "Multiprocessor Scheduling in a Hard Real-Time Environment," *Proc. Seventh Texas Conf. Comp. Syst.*, 1978.

[11] Mok, A., "Fundamental Design Problems for the Hard Real-Time Environment", MIT Ph.D. Dissertation, May 1983.

[12] Mok, A., S. Sutanthavibul, "Modeling and Scheduling of Dataflow Real-Time Systems," *Proc. IEEE Real-Time Syst. Symp.*, Dec. 1985.

[13] Mok, A., P. Amerasinghe, M. Chen, S. Sutanthavibul, K. Tantisirivat, "Synthesis of a Real-Time Message Processing System with Data-Driven Timing Constraints," *Proc. IEEE Real-Time Syst. Symp.*, Dec. 1987.

[14] Ramamritham, R., and J. A. Stankovic, "Dynamic Task Scheduling in Distributed Hard Real-Time Systems," *IEEE Software*, Vol. 1, No. 3, 1984.

[15] Sha, L., J. Lehoczky, R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *Proc. IEEE Real-Time Syst. Symp.*, Dec. 1986.

[16] Stankovic, J. A., K. Ramamritham, and S. Cheng, "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems," *IEEE Transactions on Computers*, Vol. C-34, No. 12, 1985.

[17] Stoyenko, A., "A Schedulability Analyzer for Real-Time Euclid," *Proc. IEEE Real-Time Syst. Symp.*, Dec. 1987.

[18] Zhao, W., K. Ramamritham, and J. A. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Trans. Software Eng.*, Vol. SE-13, May 1987.

[19] Zhao, W., K. Ramamritham, and J. A. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints," *IEEE Transactions on Computers,* Vol. C-36, No. 8, Aug. 1987.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION / AVAILABILITY OF REPORT approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UMIACS TR 88-15 CS TR 1991	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION University of Maryland / 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) Department of Computer Science University of Maryland College Park, MD 20742	7b. ADDRESS (City, State, and ZIP Code) 800 North Quincy Street Arlington, VA 22217-5000
8a. NAME OF FUNDING / SPONSORING ORGANIZATION / 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-87-0241
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)	
12. PERSONAL AUTHOR(S)	

13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM ______ TO ______	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT 21

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A heuristic technique for dynamically scheduling tasks in a real-time system is described in this paper. Each task is defined by a begin time, a deadline and a required computation time. The scheduling problem is to schedule every task successfully, i.e., each task should start after its begin time and complete its execution before its deadline. Two characteristics distinguish our task model and heuristic technique: First, a new task to be scheduled is allowed to modify a previous, partial schedule that is already feasible. Thus, the heuristic technique can be used either to build a feasible schedule from scratch or extend a partial, feasible schedule. Second, each task is bounded by a window of time defined by the *begin* and *end* times. The begin time is typically equal to or greater than the task arrival time. The end time is typically equal to or less than the task deadline. The additional constraint of starting a task after specified time instant is useful in task synchronization. Comparisons with known heuristic techniques for dynamically scheduling tasks in a real-time system are made via simulation. The results show that the proposed approach has a higher degree of success in obtaining a feasible schedule compared to the other approaches.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Satish K. Tripathi	22b. TELEPHONE (Include Area Code) 301-454-5165 / 22c. OFFICE SYMBOL Professor

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified